slug: the-anatomy-of-an-inspector title: How To Light Up The Dark Forest - A Walkthrough of Building A Cryptopunk MEV Inspector authors: [robert] tags: [flashbots] image: /img/transparency-january-1.jpeg hide_table_of_contents: false forum_link: https://collective.flashbots.net/t/how-to-light-up-the-dark-forest-a-walkthrough-of-building-a-cryptopunk-mev-inspector/881

A step-by-step walkthrough of how to add a new MEV inspector using a recently added cryptopunk snipers inspector as an example.

Punk snipers and mev-inspect-py

Last month I was pinged on Twitter about a novel form of MEV involving cryptopunks, one of the first and most valuable NFTs. Cryptopunks, or punks, were designed with their own native marketplace in the contract. You can list your punk, bid on other punks, and accept bids from the cryptopunk smart contract itself.

However, the design of this marketplace left open an area for frontrunning bots! When a punk owner accepted an offer to sell their punk they were actually accepting the highest existing offer on the market. If someone offered to buy a punk at a very low price and the punk owner accepted that offer, then MEV bots would race in front of the punk owner's acceptance with an offer that was just barely higher than the original low priced offer. As a result the MEV bot's offer would be the highest when the owner's acceptance was executed, and the MEV bot would receive the punk at a great price.

By doing so these sniper bots were able to systematically buy cryptopunks at low prices at the expense of the users who had originally placed low bids! The punk community was upset because the buyers who worked to find and attempted buy punks at low prices weren't being rewarded – bots were.

With this in mind I turned to mev-inspect ("Inspect") to try to shine a bright light on what was going on in the underbelly of the punk market. Inspect is Flashbots' open source tool to inspect historical Ethereum blocks for MEV extraction, and the data pipeline underlying MEV-explore. Inspect attempts to classify MEV extraction by strategy and to surface relevant data for analysis. For example, it can spot arbitrage between a number of top DEXes as well as liquidations. It is the perfect tool to analyze new MEV strategies.

Inspect had gone through a few iterations since I had last contributed. It was now 10x more professional with 10x the features because of the great work by my colleagues Taarush, Gui, and Luke. As a team we felt that the codebase was ready for more contributions, and I was excited to try to do that by adding a new inspector for this punks use case.

The cryptopunk snipe inspector is now live and in production. This blog post documents the process of writing a new inspector and uses the cryptopunk snipe inspector as a reference. I hope it serves as a motivation and guide for others to contribute to Inspect. A follow on blog post will focus on the data surrounding cryptopunk sniping MEV.

Building with mev-inspect-py

At a high level inspect works by getting all of the traces for a transaction, classifying traces, building schemas, inferring MEV strategies, and finally posting data to a database. At the end of this process inspect will have populated a database with structured data about the mev strategies it found.

To add a new MEV strategy, the cryptopunk sniping, to Inspect I took the following steps: * Add the cryptopunk ABI to our list of ABIs (link) * List classifications for the actions we want under the trace schema(link) as well as a cryptopunk entry in the protocols (link) * Create a punk classifier spec (link) * Build schemas with the data we want(link) * Parse the traces into our new punk schemas (link) * Add database models (link) and crud functions (link) * Add Alembic files for database migrations (link)

Here I implemented a new strategy inspector, which required all 7 steps.**But if you're adding a new protocol (e.g. a DEX) to an existing inspector (e.g. arbitrage), then only the first 3 steps are necessary because steps 4 to 7 have already been completed. New specs will fit into existing schemas, parsing scripts, and database integrations.**

Regardless, let's look at each of these 7 steps in detail. This process is only loosely linear and looks more intimidating than it is. Some steps are performed in parallel and you may revisit some steps as you design your objects.

Add the cryptopunk ABI

To identify a contract and its functions Inspect needs its ABI. With a quick Google search I found theCryptoPunks contract address on Etherscan and grabbed the abi by clicking "contract," scrolling down, and copy/pasting the ABI into a new file. That file was named "cryptopunks.json" and placed in a "cryptopunks" folder inside the ABI folder.

Creating Classification and Protocol entries

(This blog post assumes some knowledge of traces. Please reviewthis blog post if you need an intro.)

For a given trace Inspect will attempt to classify it by checking: * What address is this trace interacting with? * Does that address have an ABI that Inspect knows about? * Is the trace calling a function we care about?

Thus far we have only given Inspect an ABI, but soon we'll write a specification that tells Inspect what addresses and functions to look for. Whenever it finds that address and function combination in a trace it will classify that trace with a label that you give it. This step is about creating those labels.

Given we are classifying traces, we add a classification to thetraces.py schema. This is as easy as listing a short description of the things we want to classify, in this case, punk bids (punk_bid) and punk bid acceptances (punk_accept_bid).

We also add an entry under "Protocol" for the protocol we're working on if it doesn't already exist. Again, we add "cryptopunks" to the list of protocols.

Create a punk classifier spec

A classifier spec builds on the Classification and Protocol entries we just made by telling Inspect what to look for in classifying traces. For the cryptopunk sniper it looked like this: python from mev_inspect.schemas.traces import Protocol, Classification from mev_inspect.schemas.classifiers import (ClassifierSpec, Classifier,) class PunkBidAcceptanceClassifier(Classifier): @staticmethod def get_classification() -> Classification: return Classification.punk_accept_bid class PunkBidClassifier(Classifier): @staticmethod def get_classification() -> Classification: return Classification.punk_bid CRYPTO_PUNKS_SPEC = ClassifierSpec(abi_name="cryptopunks", protocol=Protocol.cryptopunks, valid_contract_addresses=["0xb47e3cd837dDF8e4c57F05d70Ab865de6e193BBB"], classifiers={ "enterBidForPunk(uint256)": PunkBidClassifier, "acceptBidForPunk(uint256,uint256)": PunkBidAcceptanceClassifier, },) CRYPTOPUNKS_CLASSIFIER_SPECS = [CRYPTO_PUNKS_SPEC]

Let's work through this. First, we create classes for the classifiers of the action that we are interested in. In our case we want to classify punk bids and acceptances, so we create PunkBidAcceptanceClassifier and PunkBidClassifier.

Inspect asks you to be specific about how these classifiers are used. The reason for this is that we want to be able to identify some types of actions across many protocols, but not all protocols are designed in the same way. For example, the function to swap assets is different across many DEXes, so in order to build a single "swap" classification, we need to tell Inspect what to do for each protocol you want to classify swaps on.

For one-off protocols, like my punk inspector, you can copy the template that I used and replace the punk-specific parts. But if you are adding support for a protocol to classify an action that isn't standardized across the ecosystem I'd suggest looking at how Swaps are implemented across a few DEXes.

Below these standardized functions are the spec itself. The spec has 4 parts: **abi_name*: the name of the abi entered earlier. Must match exactly the filename you specified. * *protocol*: the protocol specified in the previous step. * *valid_contract_addresses*: any contract addresses that you want to look for. Note that this can be omitted in the case that you don't want to limit a classifier to a set of addresses. * *classifiers*: a list of functions that you want to look for along with their classifications. For example, we want to look for punk bids, so we create an entry for the "PunkBidClassifier" for the function "enterBidForPunk(uint256)." Note that classifiers work through the usage of function selectors, so you must specify the function name and its types with no spaces.

Lastly, you must add the classifier spec to theinit.py file to make it visible to Inspect. At this point, Inspect will start to classify traces associated with your protocol :tada:!

After running inspect on blocks that contain the MEV you are looking for you canview the output of Inspect by pulling up the database and querying the "classified_traces" table for your ABI. If you are adding support for a new protocol (e.g. a dex) to an existing inspector (e.g. arbs) then your work would be finished at this stage!

Build schemas with the data we want

Inspect works by building structured objects we call schemas. Each schema is stored in a separate file in theschemas folder and is built to store data about some object. They look like this:

python class PunkBid(BaseModel): block_number: int transaction_hash: str trace_address: List[int] from_address: str punk_index: int price: int

You will need to make a schema for each new object you'd like to create. In my case, I created three: punk bids, punk bid acceptances, and punk snipes. In making a schema think about the information that you and others will want to see.

Parse the traces into our new schemas

With traces being classified for the actions we want we can start to query for our classifications and write the logic of our strategy inspector. The way to do this is simple. As an example: python if trace.classification == Classification.punk_bid: punk_bid = PunkBid(transaction_hash=trace.transaction_hash, block_number=trace.block_number, trace_address=trace.trace_address, from_address=trace.from_address, punk_index=trace.inputs["punkIndex"], price=trace.value,) return punk_bid

For every new strategy inspector, we create a file in themev_inspect folder and reference it in inspect_block.py, the heart of Inspect. The function you are calling should take in a list of classified traces and return a schema.

In the case of punk bids and acceptances, we need little logic to build our schemas. However, the use case of identifying

punk snipes requires some logic and a list of both bids and acceptances. We first get bids and acceptances, then use these as inputs to a function that checks for snipes, and returns a list of snipes if any existed.

At this point I'll add a few prints in my code to make and turn a test run to make sure everything is working as intended. You can run a test with the following command: python ./mev inspect [BLOCK_NUMBER]

Add database models and crud functions

Now almost all of the logic of your inspector has been completed. What remains is to post the data you've generated to databases for easy querying and viewing. To do this we need to create models of what our database entry should look like, and crud functions to write/delete from the database.

Database models specify the data you want to be stored and what data types each entry should be. Here's what the punk snipe entry looks like:

python class PunkSnipeModel(Base): __tablename__ = "punk_snipes" block_number = Column(Numeric, nullable=False) transaction_hash = Column(String, primary_key=True) trace_address = Column(ARRAY(Integer), primary_key=True) from_address = Column(String, nullable=False) punk_index = Column(Integer, nullable=False) min_acceptance_price = Column(Numeric, nullable=False) acceptance_price = Column(Numeric, nullable=False)

You should create a unique file and store it in themodels folder. Crud functions are stored inthe crud folder. These use our schemas and database models to write and delete from the Inspect database. I suggest you read the punk crud as an example and use it as a template to create your own by replacing the schema and models.

Add Alembic files for database migrations

Alembic is a lightweight database migration tool. We use it to manage different revisions to the inspect database. Given you're adding a new inspector you'll need to make changes to the databases.

To spin up a new alembic revision file use the following command:./mev exec alembic revision —autogenerate -m "{message}" Then copy it out of your kubernetes pod into your local machine with: kubectl cp pod:dir/file local_dir/file Now edit this file and add the features of the database that you want. This should closely resemble the model that you created in the last step. The alembic file for the punk snipe database entry is a nice example. Finally, upgrade your database with the new migration with: ./mev exec alembic upgrade head

Putting it all together

Now you are ready to inspect a block and query the database for results. First we run Inspect over a block:./mev inspect [BLOCK_NUMBER] Now we connect to the local postgres database:./mev db You should see mev_inspect=#. In my case I wanted to check out my new punk tables, so I ran SELECT COUNT(*) FROM punk_snipes; Which returned 1! That means Inspect is successfully finding punk snipes.

Congratulations! You've learned about the different moving parts involved in creating a new MEV inspector. I hope you'll give writing one yourself a shot, and don't be afraid to jump in the Flashbots Discord to ask questions in #mev-inspect. We have a list of current and desired classifiers on the Inspect repo if you suggestions for where to start. Come illuminate the dark forest with us, anon!

Thank you to Taarush, Chris, Gui, Luke, and Alex for reviewing this article.